信息内容安全 Project 设计文档

页面遍历策略的实现

陈辛楷 15300240005

Part 1

使用说明(README)

1.简介

编程语言：Python
运行环境：Python 2.7.13
集成开发环境：JetBrains PyCharm Community Edition 2016.3.2

2.使用说明

2.1 外部包安装

2.1.1 BeautifulSoup

在运行程序之前，需要先安装 BeautifulSoup，在命令行窗口中输入以下命令（针对 Windows 系统）：

```
pip install bs4
```

这一命令的前提是您已安装了 pip，它一般在安装 Python 时就已经安装在您的计算机中。若没有安装 pip，您可以通过 https://pypi.python.org/pypi/pip 下载 pip 并解压后，从命令行窗口进入解压的文件夹，输入：

```
python setup.py install
```

即可安装 pip。
Beautiful Soup 安装完成后，即可运行我们的程序了。

2.2 程序运行

2.2.1 打开程序

您可以在任何支持 Python 的 IDE 中运行本程序（如 PyCharm），也可以通过命令行窗口来实现：

通过命令行窗口进入程序所在位置，并输入：

```
python WebCrawler.py
```

即可运行本程序了。

2.2.2 输入参数

当您成功运行本程序的时候，便可以根据提示输入您希望爬取的 URL、爬取的数量和使用的算法了。

首先您会看到：

```
Welcome !

Enter URL to crawl:
```

您需要输入您希望爬取的 URL，例如 https://www.google.cn。

然后，输入您所想要爬取的网页的数量，比如 100：

```
Max Count [e.g. 100]:
```

最后，输入您所希望使用的算法，1 代表宽度优先 BFS，2 代表深度优先 DFS：

```
Choose an algorithm [1.BFS    2.DFS]:
```

接下来，我们就可以开始爬取网页了。

Part 2

程序设计

1.总体思路

1.1 使用 Python 编程

我们要实现的是一个网络爬虫，那么使用 Python 就几乎是一个很自然的想法。Python 中的许多模块都能够帮助我们来爬取网页。

比如 urllib2 模块，用于请求 URL，是一个很有效的工具。如 urllib2.urlopen 函数是用来打开 URL 网址，urllib2.request 函数用于 URL 请求等。

还有 BeautifulSoup 模块, 这并非 Python 自带的模块, 但它可以更好地帮助我们从 HTML 文件中提取数据。它能把数据解析成一个 BeautifulSoup 的类，并支持一些第三方的 HTML 解析器，如 lxml，以提高效率。比如，我们可以使用 BeautifulSoup 中的 findAll 来找到对应网页的所有 html 语句（<a href="">）。

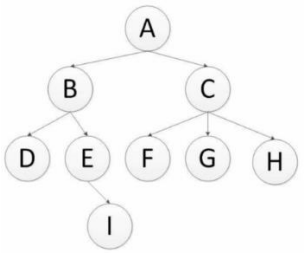
虽然老师对于这个 Project 的要求是具有一定的工作量，但这并不代表我们应当一味地追求代码量。使用 Java 自然也能够完成我们的工作，但 Python 能大大提高我们完成工作的效率。在实际的工作中，Python 的重要性也不言而喻。

1.2 宽度优先(BFS)和深度优先(DFS)

宽度优先(Breadth-First Search, BFS)和深度优先(Depth-First Search, DFS)，这是我们熟悉的名词。在学习数据结构的时候，我们曾使用这两种方法实现对树的遍历。而对于网络爬虫来说，其实也是类似的，一个个站点及其之间的联系和树十分相像，我们自然也可以以此为切入点，实现对网站的爬取。

宽度优先，是指将新下载网页发现的链接直接插入到待抓取 URL 队列的末尾，也就是指网络爬虫会先抓取起始页中的所有网页，然后在选择其中的一个连接网页，继续抓取在此网页中链接的所有网页。以下图为例，其爬取的顺序是 A-B-C-D-E-F-G-H-I。

深度优先，则是指网络爬虫会从起始页开始，一个链接接着一个链接跟踪下去，处理完这条线路之后再转入下一个起始页，继续追踪链接。以下图为例，其爬取的顺序是 A-B-D-E-I-C-F-G-H。



对于宽度优先遍历，我们自然是使用一个队列来实现，将新抓取出的 URL 放置在未访问 URL 队列的末尾，这样就很自然地实现了对先抓取的页面先访问，即先进先出。

对于深度优先遍历，我们可以使用递归来实现，若该链接还有子链接，进入那个子链接进行递归，直到不再有子链接时，返回上一层。当然，我们也可以使用堆栈来实现非递归的算法。每次都将栈顶元素取出(pop)，并将与其相连的未访问过的结点（在这里便是子链接）push 进栈，以实现我们的深度优先。

从以上过程我们可以看出，BFS 和 DFS 事实上具有高度的对称性或者说相似性，除了出列的方式不同之外，它们是如此的一致。因此，在 Python 中我们不需要将两种数据结构分离开，只需要构建一个数据结构，如本程序中的 UrlSequence，它可以为两种情况服务。在 UrlSequence 中设置的未访问序列 self.unvisited 可以完成出队列和出栈的操作，前者可以通过 pop(0)来实现，后者自然是 pop()。

在队列和堆栈的“融合”中，我们只需要一个数据结构和一个程序就可以实现两种搜素算法，不仅方便了使用者，也使得自身的工作变得更加简洁。

1.3 网页解析

在网络爬虫的实现过程中，很重要的一点就是如何解析网页，提取 URL。笔者将它分成两个过程：首先，获得网页的源代码——这使得我们能很方便地获得该网页中包含的所有链接，以便我们进行下一步的爬取；其次，自然是获取链接，再加入我们的未访问序列中。

对于第一步获取网页源代码的过程，笔者使用了之前提到的 urllib2 模块进行操作。通过 urllib2 模块的 request 等函数，实现了 HTTP 的 request 和 respond 的过程，从而获得网页的源代码。

对于第二步获取网页链接的过程，笔者使用了 BeautifulSoup 模块，对源代码中所有的链接进行访问和存储，加入到待访问的序列中，为下一步做准备。

这其中所涉及的有关 HTML 和 HTTP 报文的内容是编写程序的基础，由于笔者先前使用过 HTML 制作静态网页，以及在计算机网络课上学习了应用层的内容，对这二者还是有着基本的了解，因此这一块笔者做起来还算顺利。

2.详细设计

2.1 概述

为了实现该程序，笔者定义了两个类：第一个叫 Crawler，用于实现 URL 序列的初始化和爬取网页的过程，以及获取网页源码和链接的函数；第二个叫 UrlSequence，即实现了 URL 序列中的进、出序列，计数等操作，并维护了两个列表，一个是已访问过的序列 visited，另一个是未访问过的序列 unvisited。这两个类从字面上来看都很好理解，这提高了程序的可读性和写程序时的效率。

2.2 具体分析

2.2.1 UrlSequence 类

UrlSequence 类中存放了所有和 URL 序列有关的函数，维护了两个序列 visited 和 unvisited，并实现了对 URL 序列的元素增加、元素删除、获取元素数量等操作。

①初始化函数__init__()：

```
def __init__(self):
    # the set of visited URLs
    self.visited = []
    # the set of unvisited URLs
    self.unvisited = []
```

该函数即是对上述两个集合 visited 和 unvisited 的初始化，即是已访问过的 URL 和未访问过的 URL 的集合。笔者在程序中相应的地方都添加了英文注释，以方便读者阅读。

②获取上述序列的函数 getVisitedUrl()和 getUnvisitedUrl()：

```
# get "visited" list
def getVisitedUrl(self):
    return self.visited
```

```
# get "unvisited" list
def getUnvisitedUrl(self):
    return self.unvisited
```

这两个函数就是返回了两个序列。

③对已访问的 URL 序列的增加 Visited_Add()：

```
# Add to visited URLs
def Visited_Add(self, url):
    self.visited.append(url)
```

当我们访问完一个 URL 之后，要将它加入 visited 序列，以表示该 URL 已经被访问过了。该函数调用了 append()函数，将该 URL 加入 visited 序列的末尾。在深度优先算法中，如果

不使用 FILO 模式而是 FIFO 模式（即和宽度优先一样选择队列的出队方式）的话，则对于 visited 序列应该使用 insert 函数，即加入到队列的头部，这也能实现我们的深度优先，不过这里仅作一个讨论。

④对未访问的 URL 队列的出队 Unvisited_Dequeue()：

```
# Dequeue from unvisited URLs
def Unvisited_Dequeue(self):
    try:
        return self.unvisited.pop(0)
    except:
        return None
```

当我们需要对未访问过的 URL 进行宽度优先访问时，就需要调用该函数。我们调用了 pop()函数来实现 dequeue 的功能。需要注意的是，pop 函数在默认情况下返回队列最末尾的一个元素，因此要使用 pop(0)。

⑤对未访问的 URL 序列的出栈 Unvisited_Pop()

```
# Pop from unvisited URLs
def Unvisited_Pop(self):
    try:
        return self.unvisited.pop()
    except:
        return None
```

该函数和上一个函数几乎无异，但可以注意到它是将序列中最后一个元素弹出，即实现了出栈的功能，适用于深度优先的算法。可以看到，④⑤两个函数具有一种对称的美感。

⑥对未访问的 URL 序列的增加 Unvisited_Add()：

```
# Add new URLs
def Unvisited_Add(self, url):
    if url != "" and url not in self.visited and url not in self.unvisited:
        self.unvisited.append(url)
```

对于刚刚抓取到的 URL，我们要将它们加入未访问的 URL 序列中，不过这时，要保证它们不在 visited 和 unvisited 两个序列中。

⑦获得抓取/待抓取的 URL 数量 Visited_Count()和 Unvisited_Count()：

```
# The count of visited URLs
def Visited_Count(self):
    return len(self.visited)
```

```
# The count of unvisited URLs
def Unvisited_Count(self):
    return len(self.unvisited)
```

这两个函数也很简单，就返回了两个序列的元素个数，这使得我们能够控制抓取网页的个数。

⑧判断待访问 URL 序列是否为空的函数 UnvisitedIsEmpty()：

```
# Determine whether "unvisited" queue is empty
def UnvisitedIsEmpty(self):
    return len(self.unvisited) == 0
```

判断 len(self.unvisited)是否为 0。若待访问 URL 序列为空，则停止对网页的爬取。当然这一情况一般不会发生。

2.2.2 Crawler 类

Crawler 类中有四个函数，分别是初始化函数__init__()、爬取函数 crawling()、获取链接的函数 getLinks()，和获取网页源代码的函数 getPageSource()。

①初始化函数__init__()：

```
def __init__(self, base_url):
    # Using base_url to initialize URL queue
    self.UrlSequence = UrlSequence()
    # Add base_url to the "unvisited" list
    self.UrlSequence.Unvisited_Add(base_url)
    print "Add the base url \"%s\" to the unvisited url list" % str(self.UrlSequence.unvisited)
```

初始化 UrlSequence，然后对初始输入的网址 base_url 进行入队操作。

②获得网页源码的函数 getPageSource()：

```
# Get the page source code
def getPageSource(self, url, timeout=100, coding=None):
    try:
        socket.setdefaulttimeout(timeout)
        req = urllib2.Request(url)
        # Add HTTP header
        req.add_header('User-agent', 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)')
        response = urllib2.urlopen(req)
        if coding is None:
            coding = response.headers.getparam("charset")
        if coding is None:
            page = response.read()
        else:
            page = response.read()
            page = page.decode(coding).encode('utf-8')
        # HTTP Status Code 200 means requests are accepted by the server
        return ["200", page]
    except Exception, e:
        print str(e)
        return [str(e), None]
```

首先，访问网页的过程中常常会存在无法连接到目的网址的情况，因此，有必要设置一个超时 timeout。笔者使用 socket 库的接口设置超时时间。

之后，使用 urllib2 模块中的 Request()函数请求 URL，并为它加上 HTTP 的首部。首部的格式在笔者先前做计算机网络实验时有了解。之后，使用 urlopen()函数打开该 URL。

再往后，我们要判断编码，使用 getparam("charset")函数来做判断，并通过 read()函数去读取页面，存入 page 中。若 coding 不等于 None，那么将会对 page 进行重新编码，编码成 utf-8 的格式。

若一切正常，返回一个列表，第一个元素是 200，这代表了成功，因为 HTTP 的状态码 200 即是说明 HTTP 请求被服务器所接受了。列表中还存有之前读取出的 page，即网页的源代码。

③获取源码中的链接的函数 getLinks()：

```
# Get links from the source code
def getLinks(self, url):
    links = []
    data = self.getPageSource(url)
    if data[0] == "200":
        # Create a BeautifulSoup object
        soup = BeautifulSoup(data[1])
        # Find all HTML sentences <a href=".*">
        # .* is a regular expression meaning getting the content between quotation marks(""), i.e. the URLs
        a = soup.findAll("a", {"href": re.compile(".*")})
        for i in a:
            if i["href"].find("http://") != -1 or i["href"].find("https://") != -1:
                links.append(i["href"])
    return links
```

该函数即是从源码中获得链接，以进行下一步的爬取。在源码中，链接总是以“<a href=”开头，笔者正是利用这一点来寻找链接。

首先，创建一个 links 列表用来保存得到的链接。

接下来，调用 getPageSource()函数获取源码；这时，我们调用 BeautifulSoup 模块，对源码进行解析。findAll()函数帮助我们找到所有的以“<a href=”开头的字符串。需要指出的是，这里的“.*”是一个正则表达式(Regular Expression, RE)，它表示读取两个引号之间的内容，即我们需要的 URL。笔者也做了相应的注释。笔者通过 re.compile()函数生成一个正则表达式对象，来完成读取。

之后，判断所找到的 HTML 语句是否真的包含 URL，即是否含有 http://或”https://”。若有的话，将其加入 links 列表中。最后，返回 links。

④爬取函数 crawling()：

```
def crawling(self, base_url, max_count, flag):
    # Loop condition: the "unvisited" list is not empty & the number of visited URLs is not bigger than max_count
    while self.UrlSequence.UnvisitedIsEmpty() is False and self.UrlSequence.Visited_Count() <= max_count:
        # Dequeue or pop, according to the flag
        if flag == 1:
            visitUrl = self.UrlSequence.Unvisited_Dequeue()
        else:
            visitUrl = self.UrlSequence.Unvisited_Pop()
        print "Pop out one url \"%s\" from unvisited url list" % visitUrl
        if visitUrl in self.UrlSequence.visited or visitUrl is None or visitUrl == "":
            continue
        # Get the links
        links = self.getLinks(visitUrl)
        print "Get %d new links" % len(links)
        # Now "visitUrl" has been visited. Add it to the "visited" list
        self.UrlSequence.Visited_Add(visitUrl)
        print "Visited url count: " + str(self.UrlSequence.Visited_Count())
        # Add the unvisited URL to the "unvisited" list
        for link in links:
            self.UrlSequence.Unvisited_Add(link)
        print "%d unvisited links:" % len(self.UrlSequence.getUnvisitedUrl())
```

在 crawling()中设置了三个参数：base_url、max_count 和 flag。

循环的条件是：unvisited 序列不为空，并且当前已访问的 URL 数量不大于用户所设置的最大数量 max_count。

接下来，通过 flag 判断是使用深度优先还是宽度优先进行爬取。若 flag=1，表明用户希望使用 BFS，则当前访问的的 visitUrl 应当是通过 Unvisited 的 Dequeue 操作得到；否则，用户希望使用 DFS，则 visitUrl 应当是通过 Unvisited 的 Pop 操作得到。

然后，我们获取 visitUrl 对应网页中的所有链接，存入 links 中。将 visitUrl 存入 visited 序列表示已访问。

最后，遍历 links，将其中的链接加入 unvisited 序列中，开始下一层的循环。这样，我们就完成了这个 crawling()函数。

2.2.3 输入

根据 Part1 的使用说明，用户需要输入希望爬取的 URL、爬取网页的数量和使用的算法，分别保存到 urls、count 和 flag 中。

```
urls = raw_input('Welcome!\nEnter URL to crawl: ')  #input the URL
count = int(raw_input('Max Count [e.g. 100]: '))  #input the maximum number
flag = int(raw_input('Choose an algorithm [1.BFS  2.DFS]:'))  #indicate the preferred algorithm
```

2.2.4 main 函数

main 函数需要传入三个变量，base_url、max_count 和 flag：

```
def main(base_url, max_count, flag):
    craw = Crawler(base_url)
    craw.crawling(base_url, max_count, flag)
```

首先，使用 Crawler()对 base_url 进行初始化。
然后，调用 crawling 函数进行爬取。

最后，程序根据用户的输入开始了网页爬取。

```
if __name__ == "__main__":
    main(urls , count, flag)
    print "Mission accomplished! Thanks for using :)"
```

结束时，可以看到“Mission accomplished! Thanks for using :)”的语句。到这里，整个程序就完整地呈现了。

Part 3

结果测试

3.1 程序运行

我们分别使用 BFS 和 DFS 爬取 https://www.sina.com.cn/，爬取数量为 100。结果如下：

①BFS

```
Welcome!
Enter URL to crawl: https://www.sina.com.cn/
Max Count [e.g. 100]: 100
Choose an algorithm [1.BFS  2.DFS]:1
Add the base url "['https://www.sina.com.cn/']" to the unvisited url list
```

选取抓取过程的部分截图：

```
Pop out one url "https://passport.sinaimg.cn/html/sso/signupagreement.html" from unvisited url list
Get 12 new links
Visited url count: 29
16 unvisited links:
Pop out one url "https://passport.sinaimg.cn/html/sso/signupagreement_freemail.html" from unvisited url list
```

这是第 29 和第 30 个网页，在抓第 29 个网页时，我们可以看到，程序还获得了 12 个新的链接，但它接下来选择的是跟它处于同一深度的另一个网页，这也正是宽度优先的思想。

由于链接过多，笔者将抓取下来的页面记录保存到 Sina_BFS.txt 中。

②DFS

```
Welcome!
Enter URL to crawl: https://www.sina.com.cn/
Max Count [e.g. 100]: 100
Choose an algorithm [1.BFS  2.DFS]:2
```

选取抓取过程的部分截图：

```
Pop out one url "https://www.miaopai.com/" from unvisited url list
Get 7 new links
Visited url count: 39
6 unvisited links:
Pop out one url "https://m.miaopai.com/v/chanelLink/miaopai_yx_wap" from unvisited url list
```

如图所示，这是第 39 个和 40 个网页，当程序抓到 https://www.miaopai.com/ 时，下一个抓取的便是它的子链接。这正是深度优先的思想。

同样地，由于链接过多，笔者将抓取下来的页面记录保存到 Sina_DFS.txt 中。

我们再对谷歌首页进行爬取测试，采用 BFS，爬取 10 条，结果如下：

Welcome!

Enter URL to crawl: http://www.google.cn/

Max Count [e.g. 100]: 10

Choose an algorithm [1.BFS 2.DFS]:1

Add the base url "['http://www.google.cn/ ']" to the unvisited url list

Pop out one url "http://www.google.cn/ " from unvisited url list

Get 4 new links

Visited url count: 1

3 unvisited links:

Pop out one url "http://www.google.com.hk/webhp?hl=zh-CN&sourceid=cnhp" from

unvisited url list

Get 11 new links

Visited url count: 2

13 unvisited links:

Pop out one url "http://translate.google.cn/?sourceid=cnhp" from unvisited url list

Get 12 new links

Visited url count: 3

24 unvisited links:

Pop out one url "http://www.miibeian.gov.cn/" from unvisited url list

Get 0 new links

Visited url count: 4

23 unvisited links:

Pop out one url "https://www.google.com.hk/imghp?hl=zh-CN&tab=wi" from unvisited url list

Get 10 new links

Visited url count: 5

30 unvisited links:

Pop out one url "http://ditu.google.cn/maps?hl=zh-CN&tab=wl" from unvisited url list

Get 1 new links

Visited url count: 6

30 unvisited links:

Pop out one url "https://play.google.com/?hl=zh-CN&tab=w8" from unvisited url list

Get 16 new links

Visited url count: 7

44 unvisited links:

Pop out one url "https://www.youtube.com/?gl=HK&tab=w1" from unvisited url list

<urlopen error [Errno 10060] >

Get 0 new links

Visited url count: 8

43 unvisited links:

Pop out one url "https://news.google.com.hk/nwshp?hl=zh-CN&tab=wn" from unvisited url list

<urlopen error [Errno 10060] >

Get 0 new links

Visited url count: 9

42 unvisited links:

Pop out one url "https://mail.google.com/mail/?tab=wm" from unvisited url list

Get 7 new links

Visited url count: 10

48 unvisited links:

Pop out one url "https://drive.google.com/?tab=wo" from unvisited url list

<urlopen error [Errno 10060] >

Get 0 new links

Visited url count: 11

47 unvisited links:

Mission accomplished! Thanks for using :)

3.2 一些问题

总体来说，爬取的过程还是比较顺利的，对于各大网站的爬取也都支持。
但笔者还是发现了一些问题：

- 偶尔存在连接不到服务器的情况，但发生该情况的数量应当仍属于正常现象。
- 有时爬取一个网页要花很长的时间，在 DFS 下可以理解，但 BFS 也会存在这种情况，有可能是连接中断或连接超时。
- 对有些网页的爬取情况不理想。如百度首页，在某个时段就无法进行爬取，而笔者同时段测试了对百度贴吧主页(https://tieba.baidu.com/)的爬取，是没有问题的，此外，笔者还尝试了谷歌首页(http://www.google.cn/)、凤凰网首页(http://www.ifeng.com/)、BBC 新闻(http://www.bbc.com/news)等网页，爬取都没有问题。

Part 4

源代码

```
# encoding=utf-8
from bs4 import BeautifulSoup
import socket
import urllib2
import re

urls = raw_input('Welcome!\nEnter URL to crawl: ')    #input the URL
count = int(raw_input('Max Count [e.g. 100]: '))    #input the maximum number
flag = int(raw_input('Choose an algorithm [1.BFS    2.DFS]:'))    #indicate the preferred algorithm

class Crawler:
    def __init__(self, base_url):
        # Using base_url to initialize URL queue
        self.UrlSequence = UrlSequence()
        # Add base_url to the "unvisited" list
        self.UrlSequence.Unvisited_Add(base_url)
        print "Add the base url \"%s\" to the unvisited url list" % str(self.UrlSequence.unvisited)
```

```
    def crawling(self, base_url, max_count, flag):
        # Loop condition: the "unvisited" list is not empty & the number of visited URLs is not bigger than
max_count
        while self.UrlSequence.UnvisitedIsEmpty() is False and self.UrlSequence.Visited_Count() <=
max_count:
            # Dequeue or pop, according to the flag
            if flag == 1:    # using BFS
                visitUrl = self.UrlSequence.Unvisited_Dequeue()
            else:    # using DFS
                visitUrl = self.UrlSequence.Unvisited_Pop()
            print "Pop out one url \"%s\" from unvisited url list" % visitUrl
            if visitUrl in self.UrlSequence.visited or visitUrl is None or visitUrl == "":
                continue
            # Get the links
            links = self.getLinks(visitUrl)
            print "Get %d new links" % len(links)
            # Now "visitUrl" has been visited. Add it to the "visited" list
            self.UrlSequence.Visited_Add(visitUrl)
            print "Visited url count: " + str(self.UrlSequence.Visited_Count())
            # Add the unvisited URL to the "unvisited" list
            for link in links:
                self.UrlSequence.Unvisited_Add(link)
            print "%d unvisited links:" % len(self.UrlSequence.getUnvisitedUrl())

    # Get links from the source code
    def getLinks(self, url):
        links = []
        data = self.getPageSource(url)
        if data[0] == "200":
            # Create a BeautifulSoup object
            soup = BeautifulSoup(data[1])
            # Find all HTML sentences <a href=".*">
            # .* is a regular expression meaning getting the content between quotation marks(""), i.e. the URLs
            a = soup.findAll("a", {"href": re.compile(".*")})
            for i in a:
                if i["href"].find("http://") != -1 or i["href"].find("https://") != -1:
                    links.append(i["href"])
        return links

    # Get the page source code
    def getPageSource(self, url, timeout=100, coding=None):
        try:
            socket.setdefaulttimeout(timeout)
            req = urllib2.Request(url)
```

```
                # Add HTTP header
                req.add_header('User-agent', 'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)')
                response = urllib2.urlopen(req)
                if coding is None:
                    coding = response.headers.getparam("charset")
                if coding is None:
                    page = response.read()
                else:
                    page = response.read()
                    page = page.decode(coding).encode('utf-8')
                # HTTP Status Code 200 means requests are accepted by the server
                return ["200", page]
            except Exception, e:
                print str(e)
                return [str(e), None]


class UrlSequence:
    def __init__(self):
        # the set of visited URLs
        self.visited = []
        # the set of unvisited URLs
        self.unvisited = []

    # get "visited" list
    def getVisitedUrl(self):
        return self.visited

    # get "unvisited" list
    def getUnvisitedUrl(self):
        return self.unvisited

    # Add to visited URLs
    def Visited_Add(self, url):
        self.visited.append(url)

    # Remove from visited URLs
    def Visited_Remove(self, url):
        self.visited.remove(url)

    # Dequeue from unvisited URLs
    def Unvisited_Dequeue(self):
        try:
            return self.unvisited.pop(0)
```

```
            except:
                return None

        # Pop from unvisited URLs
        def Unvisited_Pop(self):
            try:
                return self.unvisited.pop()
            except:
                return None

        # Add new URLs
        def Unvisited_Add(self, url):
            if url != "" and url not in self.visited and url not in self.unvisited:
                self.unvisited.append(url)

        # The count of visited URLs
        def Visited_Count(self):
            return len(self.visited)

        # The count of unvisited URLs
        def Unvisited_Count(self):
            return len(self.unvisited)

        # Determine whether "unvisited" queue is empty
        def UnvisitedIsEmpty(self):
            return len(self.unvisited) == 0


def main(base_url, max_count, flag):
        craw = Crawler(base_url)
        craw.crawling(base_url, max_count, flag)


if __name__ == "__main__":
        main(urls , count, flag)
        print "Mission accomplished! Thanks for using :)"
```

Part 5

感想和总结

经过这次 Project 的考验，笔者对于 Python 的编写更加熟悉，而对于网络爬虫，虽是第一次尝试，但也受益匪浅。笔者的程序还是一个比较幼稚的作品，但对笔者来说，这是一个

全新的体验。在如何获取 URL 以及两种算法的实现上，笔者曾经遇到了不少问题，也查阅了很多资料，最终克服了这些困难，感受到了爬虫的魅力。

同时，很高兴这学期能选“信息内容安全”这门课，也十分感谢曾老师，让笔者能够有机会了解这个领域的知识和技能，并完成自己的作品。

Part 6

参考资料

[1] https://stackoverflow.com/
[2] https://github.com/
[3] http://blog.csdn.net/
[4] https://www.crummy.com/software/BeautifulSoup/
[5] https://docs.python.org/2.7/library/urllib2.html
[6] https://docs.python.org/2.7/library/re.html
[7] https://docs.python.org/2.7/library/socket.html
[8] Lawrence L. Larmore(UNLV), DFS and BFS Algorithms using Stacks and Queues
[9] Ryan Mitchell, Web Scraping with Python: Collecting Data from the Modern Web